Echo Technology (ET) for Memory Constrained CISC Processors

[1]Youfeng Wu, [1]Mauricio Breternitz Jr, [2]Herbert Hum, [3]Ramesh Peri, [2]Jay Pickett
[1]Programming Systems Lab [2]Low Power Microprocessor Lab [3]Compiler Lab
Intel Labs
2200 Mission College Blvd
Santa Clara, CA 95054
{youfeng.wu,mauricio.breternitz.jr,herbert.hum,ramesh.peri,jay.pickett}@intel.com

Abstract Code density is an important issue in memory constrained processors. Echo Technology (ET) can be employed to replace a repeating code sequence with a single ECHO instruction. Its usefulness for code size reduction has been demonstrated on JAVA bytecode and RISC binaries. In this paper, we evaluate its effectiveness for IA32 binaries. Common perception assumes that IA32 code is already relatively dense due to its variable length instruction set, however, different forms of ECHO instructions of varying lengths can be used to further compress IA32 code. Our experiments show that IA32 equipped with ET is capable of achieving a similar code density as the THUMB extension in the ARM instruction set and with significantly lower performance penalty.

1 Introduction

Code density is an important issue in memory constrained processors. RISC processors such as ARM address this issue by introducing a special subset of instructions such as THUMB and THUMB2 that specifically target higher code density. This requires a significant ISA change and also extra hardware decoding overhead for handling ARM, THUMB and THUMB2 instructions. Studies also show a significant performance loss when the THUMB code is generated for code size reduction ([Nanja-04][ARM-04]).

Echo Technology (ET) replaces a repeating code sequence with a single ECHO instruction to reduce the code size for programs running on memory constrained processors. Its usefulness for code size reduction has been demonstrated on JAVA bytecode and Alpha code [Fraser-02, Lau-03]. In this paper, we evaluate its effectiveness for IA32 binaries.

IA32 processors have emerged successfully in the embedded environment [Intel-04, NS-04, VIA-04] due to its tremendously large existing code base and higher code density than RISC processors. Although IA32 by nature is less compressible, its variable length instruction set permits variable ECHO instruction lengths to improve compression ability. Our experiment shows that IA32 equipped with ET is capable of achieving a similar code density as the THUMB extension in the ARM instruction set and with significantly lower performance penalty.

The contributions of the paper are summarized as follows:

- We provide an updated study of the code density of IA32 v.s. ARM/THUMB. On the average, IA32 code optimized for size is about 23% smaller than size-optimized ARM code and about 18% to 23% larger than THUMB.
- We demonstrate that ET can reduce IA32 code size by 17% to 20%. This brings IA32 code to similar code density as THUMB code. Since THUMB often suffers serious performance loss compared to ARM code and we show that IA32 with ET incurs much smaller performance loss, IA32 with ET presents a significant performance advantage over THUMB. Although mixed mode code generation [Krish-02] and the THUMB2 extension [ARM-04] have been recently proposed to alleviate the performance penalty of THUMB, we believe these new techniques will also help size-optimized IA32 binaries to improve performance. .
- We propose new IA32 Echo instructions and describe an ET algorithm to achieve the above results. We also propose several ET extensions, such as boosted Echo, that can further improve the IA32 code density.

The rest of the paper is organized as follows. Section 2 outlines related work. Section 3 compares the code density and performance of IA32 vs. ARM/THUMB. Section 4 defines Echo technology and illustrates how it can reduce code size. Section 5 describes our ET algorithm for the experiments. Section 6 proposes new IA32 Echo instructions and presents the evaluation results. Section 0 discusses future work. Section 8 concludes the paper.

2 Related work

There have been several efforts to reduce code size through the use of classical compiler optimizations such as strength reduction, dead code elimination, and common sub-expression elimination [Cooper-99, Debray-02]. The compiler optimization called Procedural Abstraction [Liao-96, Kunc-99] is also shown to reduce code size.

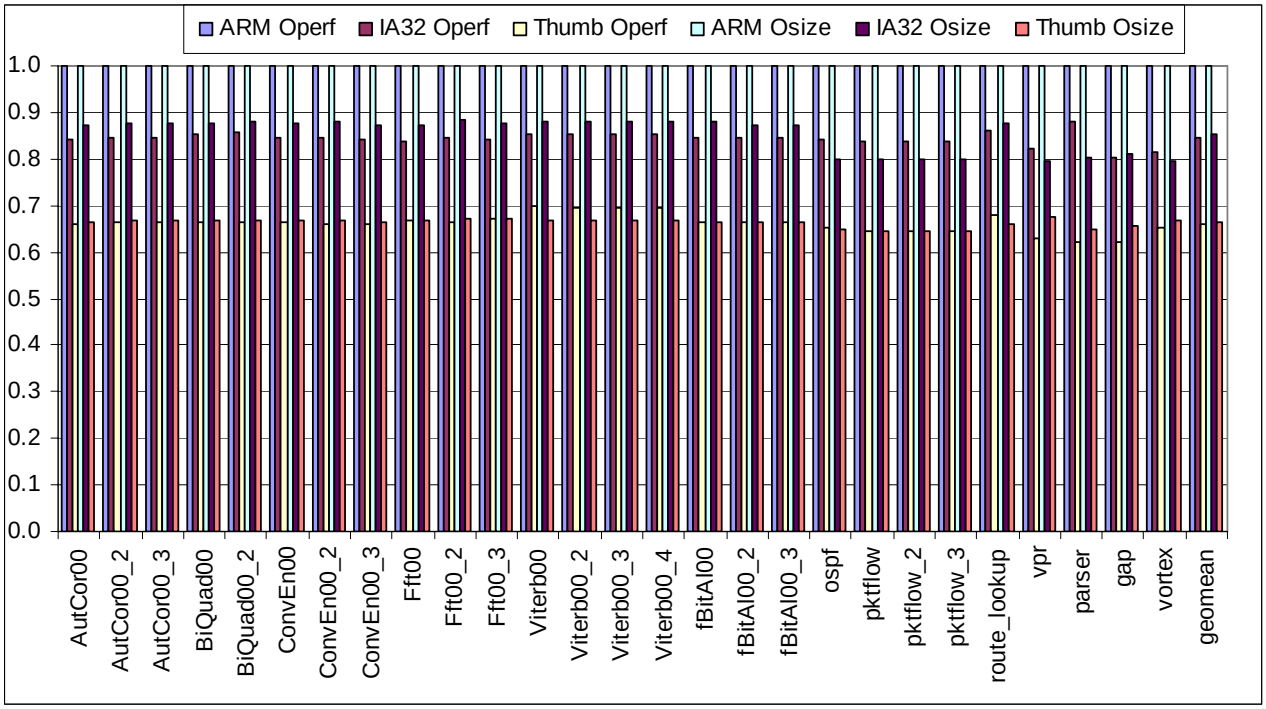

Figure 1. ARM, IA32, THUMB code size comparisons

Fraser et al [Fraser-02] first proposed the "Echo" instruction to compress Java bytecodes. They use a form of "sequential Echo" instruction, allowing calls and returns inside Echo regions. The Echo instructions reduced the size of bytecode by 30%.

Lau et al [Lau-03] extended Echo technology to include a bitmask Echo instruction. They applied the technique to the Alpha machine code without allowing call and returns inside Echo regions. With sophisticated binary rewriting techniques, including instruction reordering and register renaming, they demonstrated about 15% reduction in code size.

Brisk et al [Brisk-04] reported an early result on a framework that recognizes Echo opportunities at the compiler's intermediate representation level and asks later compiler phases to preserve the Echo opportunities discovered in earlier phases. They reported that the potential compression ratios range from 72% to 50% for 10 MediaBench applications.

3 Code density and performance

The IA32 ISA supports variable length instructions and CISC instructions. This inherently enables higher code density. For example, the GCC Code-Size Benchmark Environment [CSIBE-04] reports that IA32 code is about 10% smaller than ARM although it is about 15% larger than THUMB.

To compare the code density of IA32 to ARM/THUMB with the latest compilation technologies, we experiment with the EEMBC and SPEC2000INT benchmarks compiled with Intel's production compilers for Xscale (an ARM compatible microprocessor family) and for IA32. The two compilers are developed from the same code base and should represent the similar optimization technology and policies. Specifically, the code optimized for performance, denoted Operf, is compiled with the same optimization options (-O3 and -Qip) in both compilers, and that compiled for minimal code size, denoted Osize, is generated with the same flag (-O1) in both compilers.

Figure 1 shows the relative code sizes from our study. The ARM Operf and ARM Osize are the bases, normalized to 1. IA32 Operf and IA32 Osize are relative to the bases, respectively. Similarly, THUMB Operf and THUMB Osize are relative to the bases, respectively. For the code optimized for performance, on the average, IA32 is about 84% of ARM size, while THUMB is about 66% of ARM size. For the code optimized for size, on the average, IA32 size is 85% of ARM size, while THUMB size is about 66% of ARM size. This comparison only includes four of the SPEC2000INT benchmarks due to issues with getting all the SPEC2000INT benchmarks to work with the Xscale compiler or running them on Xscale machines (e.g. issues with memory, OS, and libraries). Still, we can see that the code size comparison of the SPEC2000INT benchmarks is similar to that of the EEMBC benchmarks.

Figure 2 focuses on the comparisons of the average code sizes for the benchmark suites. When compared

with THUMB, which is a specifically designed variant of ARM for minimal code size, the IA32 program is bigger. For example, when compiled for performance, the IA32 code is about 21% (EEMBC) to 24% (SPEC2KINT) larger than THUMB code (see the bars marked "Thumb Operf/IA32 Operf"). When compiled for code size, the IA32 code is about 18% (SPEC2KINT) to 23% (EEMBC) larger than THUMB code (see the bars marked "Thumb Osize/IA32 Osize").

Figure 2 also shows that the code size reduction from ARM Operf to THUMB Osize (about 55%) is bigger than that from IA32 compiled for performance to that for code size (37%). This is not surprising as IA32 ISA has no ISA features specifically designed for reducing code size while THUMB is exclusively designed for minimum code size. With code size of embedded IA32 applications increasing where memory space becomes more constrained, we would like to see a technology that can reduce the size of IA32 code that is already compiled for size by up to 20% so to bring its code density to similar to that of THUMB code.

Echo Technology has been shown in [Lau-03] to reduce the code size of ALPHA binaries by 15%. This code size reduction is achieved on binary that is already optimized for code size with traditional compiler techniques. In this paper, one of our proposed ET extensions (ECHO.I) achieves 17% to 20% code size reduction for IA32. Therefore, IA32 with ET should give similar code density as THUMB.

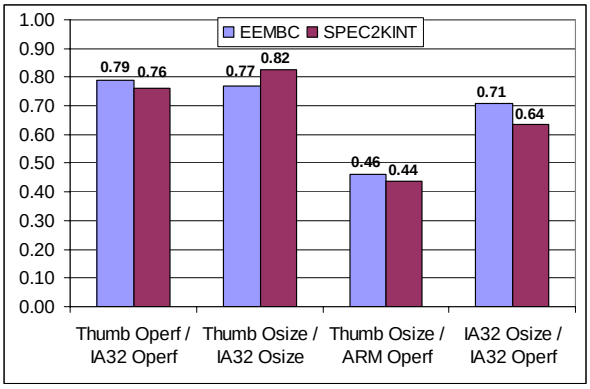

Figure 2. Code size comparison of IA32 and ARM/THUMB code (averages)

Notice that, however, the significant code size reduction from ARM to THUMB is at the cost of severe performance loss. Available data indicates that the performance loss ranges from 25% [ARM-04] to 47% [Nanja-04]. Figure 3 shows our performance measurement with Intel's Xscale compiler for the EEMBC benchmark suites. On the average, THUMB code is about 35% slower than ARM Operf code.

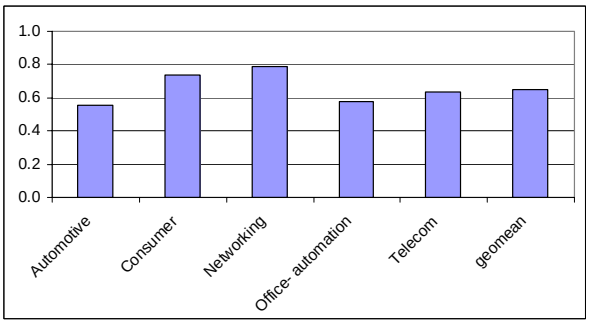

Figure 3. Performance of ARM Operf v.s. THUMB

In contrast, the performance loss from IA32 Operf to IA32 Osize is much smaller. For example, Figure 4 shows our experiment result comparing the performance of IA32 Operf to IA32 Osize binaries. On the average, there is only about 14% performance loss for the whole SPEC2000INT benchmarks.

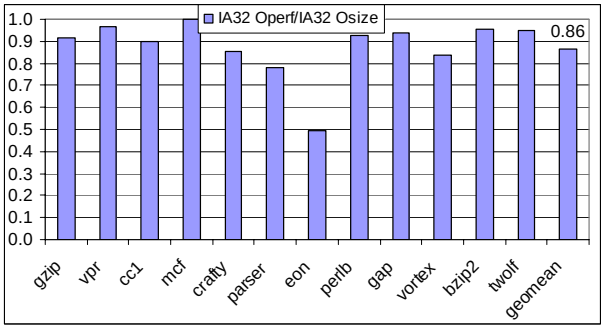

Figure 4. Performance of IA32 Operf/Osize

Lau et al reports in [Lau-03] that Echo technology has only minor performance overhead (1% to 3%). As our experiment setting is similar to [Lau-03], we expect that IA32 Echo should incur similar overhead. Therefore, IA32 optimized for code size with ET should run no more than 17% slower than IA32 optimized for performance.

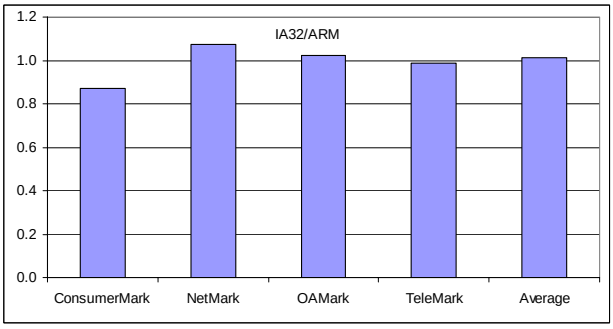

Figure 5. Relative performance of IA32 andARM

Figure 5 shows the preliminary result of a simulated IA32 processor scaled to a similar ARM processor in terms of process technology, power consumption, and die area. The two processors perform similarly. Based on this result and assume that IA32 and ARM have similar performance in general when optimized for performance, we may expect that IA32 programs

optimized for size with ET will have similar code density as THUMB code, and run significantly faster.

4 Echo Technology

Echo Technology tries to replace a repeating sequence of instructions by a single Echo instruction. An Echo instruction has the format "Echo (offset, length)". When this instruction is executed, the control is transferred to the instruction that is "offset" away from the current instruction, execute "length" instructions there, and then branch back to the instruction immediately after the Echo instruction.

For example, in Figure 6 (a), the instructions 340 to 356 are the same repeating code sequence as instructions 100 to 116. Consequently, the code sequence {340, …, 356} can be replaced by an Echo instruction, such as Echo(240, 5) shown in Figure 6 (b), indicating that when the Echo instruction is executed, the five instructions starting at 100 are executed and then execution continues at instruction 344. The value 240 is the offset from positions 100 to 340.

100	mov
104	shl
108	xor
112	add
116	movsx
	…
340	mov
344	shl
348	xor
352	add
356	movsx
	…
404	mov
408	shl
412	xor
416	add
420	movsx

(a) Original code

100	mov
104	shl
108	xor
112	add
116	movsx
	…
340	Echo(240, 5)
344	……
	…
380	Echo(280, 5)

(b) Code after applying ET

Figure 6. Echo example

For ease of discussion, we will refer to the repeating code sequence replaced by an Echo instruction the "Echo Region", and the code executed when an Echo instruction is executed is called the "Echo Target". Using this terminology, Echo Technology involves identifying Echo regions, replacing them with Echo instructions, and executing the Echo targets when the Echo instructions are executed. For the example in Figure 6, the code sequence {340, …, 356} is an Echo region, and instruction sequence {100, …, 116} is the Echo target for the Echo region. For ease of implementation, Echo targets are not allowed to contain Echo instructions (no nested Echo instructions). Notice that, Echo targets for different Echo instructions may overlap, and in fact that happens commonly. For example, it is possible that instructions {104,…, 112} may be the Echo targets of another Echo region, even though {100, …, 116} is already an Echo target.

There are restrictions on the Echo regions. First, Echo regions are not allowed to overlap. Otherwise, it will be impossible to replace the overlapping Echo regions with Echo instructions. Also, except for the first instruction, none of the instructions in a Echo region should be a target of branch instruction, as the whole Echo region will be replaced by an Echo instruction and the branch targets will no longer exist afterward. Furthermore, internal control flow inside an Echo region, such as looping or branching to internal blocks, makes the semantics of an Echo instruction hard to define and is usually not allowed.

Branch instructions with target instructions outside the Echo region may be allowed inside Echo region as long as the branches in the Echo region and the Echo target have the same target address (notice that we allow them to have different offsets in the instruction encoding). Similarly, call and return instructions with the same target addresses in the Echo region and in the Echo target may be allowed inside Echo regions.

Based on the above discussion, the semantic of an Echo instruction "Echo (offset, length)" can be specified as follows.

```
Return_PC = current PC + sizeof Echo instruction
PC = PC - offset;
Echo_Counter = length;
Echo_Mode = Ture
Continous execution instruction at PC and after each
instruction is executed, the following is checked:
If (Echo_Mode)
    If (the instruction is a branch and is taken)
        Echo_Mode = False
        Control transfers to branch target
    Else
        Echo_Counter--
        If (Echo_Counter == 0)
            Echo_Mode = False
            Control transfers to Return_PC
```

In terms of hardware implementation, a register (Return_PC) for storing the return PC and another register (Echo_Counter) for storing the length are needed. Also, a mode flag (Echo_Mode) indicating whether or not the current execution is an Echo target is necessary so the Return_PC and Echo_Counter are used/updated only when Echo target is executed. Furthermore, when call and return instructions are included inside the Echo regions, the Return_PC, Echo_Counter, and Echo_Mode need to be saved on calls and restored on returns. The save and restore can be implicitly performed by the call/return instructions on the stack.

5 ET algorithm for IA32 binaries

We will focus on reducing the code size of existing IA32 binaries via a binary rewriting tool (the same technique can be used in a link-time compilation phase). For a given IA32 binary, the tool first decodes the binary to identify the opcode and branch targets, mark those instructions with opcode that should not be included in any Echo regions (such as "int", etc), and also mark those instructions that are the targets of branch instructions. This preprocessing of the binary generates an array of the following data structure.

```
struct ProcessedInst {
        char size;
        char *abs_bits;
        char is_br_target;
        char is_non_echoble;
        char is_echo;
       unsigned compressed_pc;
};
```

The size field remembers the number of bytes the instruction occupies. The abs_bits field is a pointer to the decoded bits with the PC-relative addresses replaced by absolute addresses. The is_br_target field notes whether or not the current instruction is a branch target. The is_non_echoble field indicates that the current instruction is not allowed in an Echo region. The is_echo field is set when the current instruction is the header of an Echo region and the Echo region is replaced by an Echo instruction. When this happens, the size field will reflect the size the Echo instruction and the rest of the instructions in the Echo region are converted to non-echoble instructions with size of 0 byte. Finally, the compressed_pc is the relative PC (with the compressed_pc of the first instruction in the array being 0) of the current instruction after the previous Echo regions have been replaced with Echo instructions. This field is used to determine the offset from the Echo region to its Echo target so the appropriate Echo instruction can be used.

We assume the IA32 ISA supports K Echo instructions with different ranges for the offset and length fields $Echo_i(offset_i, length_i)$, for i = 1, …, K. The size of the $Echo_i$ instruction is Si. If the $offset_i$ occupies Oi bits in the $Echo_i$ instruction, then the maximum offset from an Echo region to its Echo target is $max\_offset_i = 2^{\wedge\wedge}Oi - 1$. If the $length_i$ field occupies L_i bits in the Echo instruction, then the maximum number of instructions in the Echo region replaced by the $Echo_i$ instruction is $max\_length_i = 2^{\wedge\wedge}L_i$. For example, for an $Echo_i$ instruction with Oi = 12 and L_i = 4, max_offset_i = 4095 and max_length_i = 16 ($length_i$ =0 implies that the Echo region has one instruction).

We use a sequential search algorithm to identify Echo regions and replace them with Echo instructions. This algorithm is based on the well-known LZ77 algorithm [Ziv-77] extended for identifying legal Echo regions in binary programs. The algorithm examines each instruction in their decoded order and tries to form an Echo region starting at this instruction with a corresponding Echo target in the instructions preceding the current instruction. An illustration of the algorithm is shown in Figure 7. The instructions 1, 2, …13, are examined in that order. When instruction 5 is examined, the instructions 1 to 4 are checked to look for an Echo target for the potential Echo region starting at instruction 5. In this example, Echo region {5, 6} is identified to match with the Echo target {1, 2}, and then the Echo region {5, 6} is replaced by Echo(4, 2). Although in the original code {9, 10} may match {4, 5} , since {5, 6} is already replaced by an Echo instruction, {9, 10} must not be identified as an Echo region for Echo target {4, 5}. Finally, Echo region {11, 12, 13} matched Echo target {2, 3, 4}, and the region can be replaced by an Echo(offset, 3) instruction, where offset is the distance from the Echo to the instruction #1, taking onto consideration of all the Echo transformations happened between them. Notice that Echo targets {1, 2} and {2, 3, 4} overlap and that presents no problem. This is one of the advantages of Echo Technology over procedural abstraction. Procedural abstraction converts common code sequences into separate procedures and uses the normal function call/ret to invoke the procedures. [Kunc-99]. The code in different procedures cannot overlap.

```
1   cmp
2   beq
3   add
4   mov
5   cmp     Echo region {5, 6} matched Echo target {1, 2},
            and replaced by Echo(4, 2)
6   beq
....
9   mov     {9, 10} could match {4, 5}, BUT {5, 6} already
             replaced by an Echo instruction.
10  cmp
....

11   beq    Echo region {11, 12, 13} matched Echo target
            {2, 3, 4}, and replaced by Echo(offset, 3)
12   add
13   mov
....
```

Figure 7. Illustation of ET algorithm

The ET algorithm must make sure that the Echo instructions used to replace the Echo regions are legal instructions supported by the architecture. In particular, when an Echo instruction Echo(offset, length) is used to replace an Echo region, there must be an $Echo_i$($offset_i$, $length_i$) supported by the hardware such that offset <= max_offset_i and length <= max_length_i. In our algorithm we guarantee this by looking for a match only among the instructions whose "compressed_pc" is within MO = {MAX(max_offset_i),

i = 1, …, k} away from the current instruction being examine, and only identify Echo regions that are no larger than ML = {MAX(max_length$_i$), for i=1, …, k}. If an Echo target is found that is less than MO away from the Echo region and/or the size of the Echo region is smaller than ML, we will select the smallest Echo instruction to replace the Echo region.

```
Struct ProcessedInst code[num_insts];

for (i=1;i<num_insts; i+=stepped_insts) {
      stepped_insts = 1;
      code[i].compressed_pc = code[i-1].compressed_pc +
                  code[i-1].size
      if (is_non_echoble_inst(i))
            continue;
      early_index = get_earlest_inst(i);
      for (unsigned int j=early_index;j<i;j++) {
            region_bytes = 0;
            region_insts = 0;
            target_idx = 0;

            if (!is_inst_match(j, i))
                  continue;
            nbytes = code[i].size;
            ninsts = 1;
            nexti = i+1;
            nextj = j+1;
            while (ninsts <max_echo_region_size &&nexti < i
                  & & (is_inst_match(nexti, nextj))) {
                  if (is_br_target(nextj))
                        break;
                  nbytes += code[nexti].size;
                  ninsts++;
                  nexti++;
                  nextj++;
            }
            if (is_better_candidate(region_bytes, target_idx,
                                    nbytes, j)) {
                  region_bytes = nbytes;
                  region_insts = ninsts;
                  target_idx = j;
            }
      }
      if (region_may_not_be_echoed(region_bytes,
            region_insts, i - target_idx, &best_echo_inst_size))
            continue;
      replace_region_by_echo_inst(i, i+region_insts-1,
            best_echo_inst_size);
      stepped_insts = region_insts;
}
```

Figure 8. Pseudo-code for ET algorithm

The detailed ET algorithm is shown in Figure 8. The input to the algorithm is the list of ProcessedInst stored in an array called "code". The i-loop sequentially examines the code array, and the j-loop tries to form an Echo region starting at code[i] with the Echo target in code[0] to code[i-1]. The function is_non_echoble_inst(i) checks that the flag code[i].is_non_echoble is set. The routine get_earlest_inst(i) finds the earliest instruction index, early_index, such that the difference between the code[i].compressed_pc and code[early_index].compressed_pc is <= MO. The function is_inst_match(j, i) compares code[j].abs_bits and code[i].abs_bits for a match. The function is_better_candidate(prev_region_size, prev_target, cur_region_size, cur_target) determines whether the current Echo region starting at instruction i is better than the previous best Echo region starting at the same instruction. Function region_may_not_be_echoed (region_size, region_inst, offset, &best_echo_inst_size) decides whether or not the candidate Echo region can be beneficially Echoed, taken into consideration of the sizes of the Echo instruction and the Echo region. If so, the smallest Echo instruction that can replace this Echo region is selected and the Echo instruction size is returned in the variable best_echo_inst_size. A unechoble region could be too small such that replacing it with any Echo instruction may not reduce code size. Finally, the routine replace_region_by_echo_inst(region_begin, region_end, echo_inst_size) replaces the current Echo region by the selected Echo instruction and also update the compressed_pc field for the instructions in the Echo region.

6 Experimental results

We experiment Echo Technology on the EEMBC1.1 and SPEC2000INT benchmark suites. We measure the effect of ET on code size reduction using:

$$\text{Compression ratio} = \frac{\text{code size with ET}}{\text{code size without ET}}.$$

All the programs are optimized for minimum code size using traditional compilation techniques before they are processed by the ET algorithm.

0F 04	37	AAA	ASCII adjust after add
0F 05	D5	AAD	ASCII adjust after divide
0F 07	D4	AAM	ASCII adjust after multiply
0F 0A	3F	AAS	ASCII adjust after subtract
0F 0C	27	DAA	Decimal adjust after add
0F 0D	2F	DAS	Decimal adjust after subtract
(a)			(b)

Figure 9. IA32 2-byte and 1-byte opcodes: (a) Available 2-byte opcodes, (b) 1-byte opcodes that might be available

6.1 IA32 Echo instructions

The IA32 instruction set has used up almost all of the 1-byte opcodes. Any new instruction may need to have at least a 2-byte opcode. Figure 9 (a) shows a few 2-byte opcodes available, which may be used for new Echo instructions. Furthermore, when IA32 processors are used in a memory constrained embedded environment, some of the existing instructions may no be longer needed. This would free up a few one-byte opcodes. For examples, some of the ASCII manipulation instructions have one-byte opcodes (see Figure 9 (b)). We potentially can retire them in the

embedded environment to allow short Echo instructions.

We propose two alternative Echo instruction extensions to IA32, as shown in Figure 10. The instructions in Figure 10 (a) assume four 1-byte opcodes are available and we can use them to support four new Echo instructions which we will be referred as Echo.I instructions. We use the notation Echo.I.operand_size.length_size to name the Echo.I instructions. For example, Echo.I.1.0 uses 1-byte to encode the two operands and the length field uses 0 bit (with a default length of 1). Therefore, the offset field uses 8 bits for a maximum offset of 255. Similarly, Echo.1.1 uses 1-byte to encode the two operands and the offset field uses 7 bits (for a maximum offset of 127) and the length field uses 1 bit (for a maximum of 2 instructions per Echo region). It is easy to see that the Echo.I instructions in Figure 10 (a) will be 2, 2, 3, and 4 bytes long, respectively.

The instructions in Figure 10 (b) assume four 2-byte opcodes are available and we can use them to support four new Echo instructions which we will refer to as Echo.II instructions. These Echo.II instructions take 1, 1, 2, or 3 bytes of operands. The four Echo instructions in Figure 10 (b) will be a total of 3, 3, 4, or 5 bytes long, respectively. We will experiment both the Echo.I and Echo.II extensions to see their relative effects on code size reduction.

Echo.I.1.0	8-bit-offset, 0-bit-counter
Echo.I.1.1	7-bit-offset, 1-bit-counter
Echo.I.2.2	14-bit-offset, 2-bit-counter
Echo.I.3.4	20-bit-offset, 4-bit-counter
(a) Echo instructions with 1-byte opcodes	
Echo.II.1.0	8-bit-offset, 0-bit-counter
Echo.II.1.1	7-bit-offset, 1-bit-counter
Echo.II.2.2	14-bit-offset, 2-bit-counter
Echo.II.3.4	20-bit-offset, 4-bit-counter
(b) Echo instructions with 2-byte opcodes	

Figure 10. Two proposed IA32 extensions for Echo instructions

6.2 Effects of the Echo instructions

Figure 11 shows code size reduction with Echo.I and Echo.II for EEMBC and SPEC2000INT benchmarks. On the average, Echo.I instructions may reduce code size by about 17% and 20% for EEMBC and SPEC2000INT, respectively, and Echo.II instructions may reduce code size by about 12% and 14% for EEMBC and SPEC2000INT. The SPEC2000INT benchmarks have slightly better compression, probably because that SPEC2000INT are larger binaries and have more repeated code.

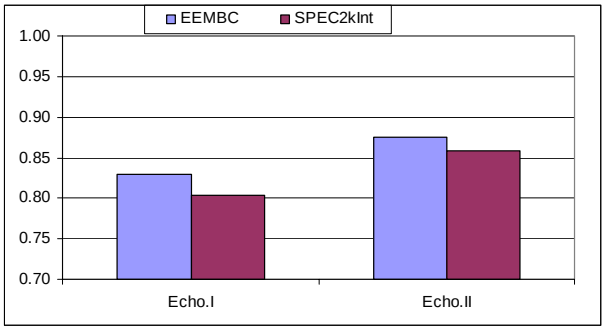

Figure 11. Compression ratios with Echo.I and Echo.II instructions

Figure 12 shows compression ratios for individual SPEC2000INT benchmarks. The code size reduction is relatively consistent, ranging from 15% to 23% for Echo.I and 10% to 17% for Echo.II.

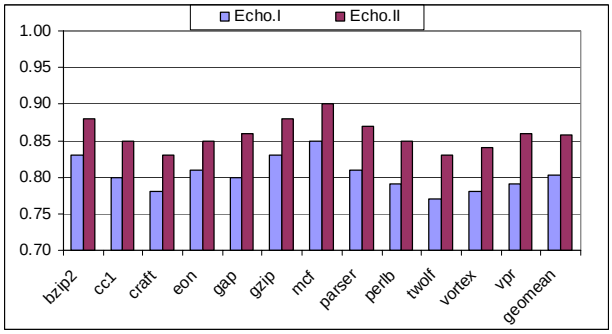

Figure 12. Compression ratios for individual SPEC2000INT benchmarks

Figure 13 shows the percentage of Echo regions that are replaced by each of the four Echo instructions. All the 4 Echo instructions in each of the two extensions are useful, although Echo.I.1.1 is the least useful for SPEC2000INT benchmarks, and Echo.I.3.4 is least frequently used for EEMBC benchmarks.

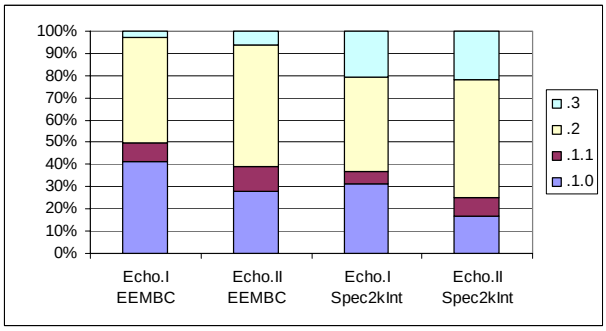

Figure 13. Relative usefulness of Echo Instructions

Figure 14 shows the distribution of the Echo region sizes (number of bytes) for SPEC2000INT benchmarks. Although there are Echo regions with more than 60 bytes, majority of the Echo regions are relatively small, with about 3 to 13 bytes.

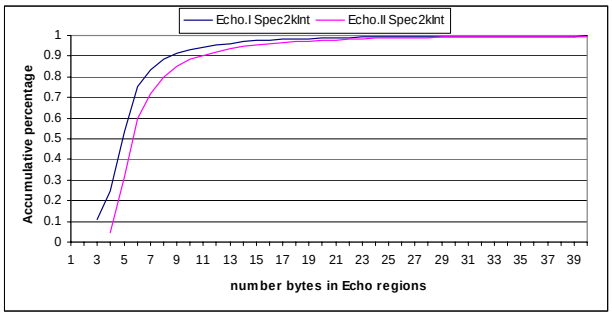

Figure 14. Distribution of Echo region sizes (number of bytes)

Figure 15 shows the distribution of the number of instructions in Echo regions for SPEC2000INT. Majority of the Echo regions have 1 to 6 instructions.

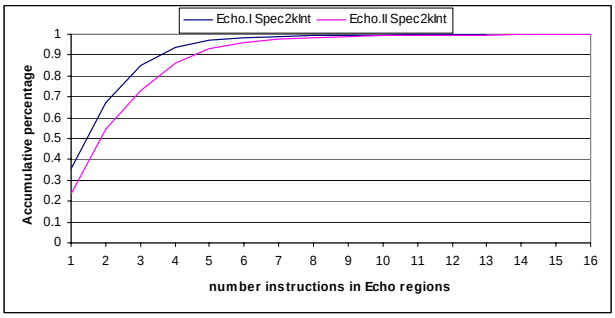

Figure 15. Distribution of number of instructions in Echo regions

In the above experiments, we have allowed Echo regions to include call, return, and branch instructions that have the same targets as their counterparts in the Echo target. Figure 16 shows the result of Echo.I with EEMBC and SPEC2000INT benchmarks when call/return, and branch instructions are selectively allowed in Echo regions. When branches are disallowed inside Echo regions, the compression ratio gets slightly worse from 83% to about 84% for EEMBC and from 80% to 82% for SPEC2000INT (See the bars marked with "Call/ret only"). When branches are allowed but call/returns are disallowed, the compression ratio gets worse to 86% for EEMBC and 85% for SPEC2000INT (see the bar marked with "Br only"). Furthermore, if all call, return, and branches are disallowed, the compression ratio further degrades to 88% for EEMBC and 86% for SPEC2000INT (see the bars marked with "No call/ret/br"). This result suggests that allowing call/return inside Echo region is very helpful, reducing code size more significantly than allowing branches. Allowing branches with the same target addresses in the Echo region is relatively easy to implement. Allowing call and return inside Echo regions may require more hardware to save the Echo registers (Return_PC, Echo_Counter, and Echo_Mode) on function call and restore them on return. Therefore, depending on the hardware budget, we have 4 possible implementations to support ET. Even with the least costly HW alternative ("No call/ret/br"), we can expect to see 12% to 14% code size reduction with Echo.I.

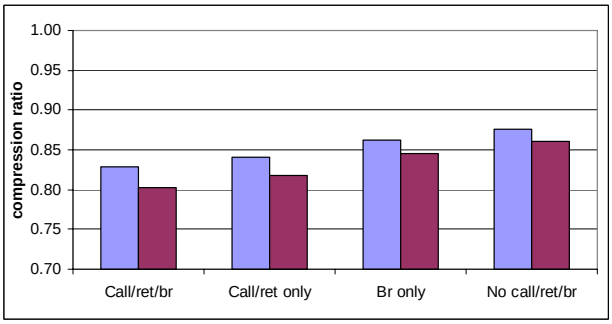

Figure 16. Compression ratios of selectively allowing call/return/branches in Echo regions

6.3 Discussions

The above results show that our ET algorithm can reduce IA32 code size by 17% to 20% if a 1-byte opcode is available, or 12% to 14% by using the available 2-byte opcodes. If calls and returns are disallowed in Echo regions, the code size reduction can still be around 14% to 15% for SPEC2000INT benchmarks with Echo.I. In contrast, a similar amount of code size reduction for Alpha is achieved in [Lau-03] with a much more complicated algorithm that requires instruction reordering, register renaming, and a new type bitmask Echo instruction. We believe that if we add Lau's techniques into our algorithm, we would significantly improve our results. This demonstrates the advantage of IA32's variable length instructions. When the offset and length fields for an Echo region can be fit into one byte (or two bytes), we can use a two-byte (or three byte) Echo instruction to replace the Echo region in IA32. The same Echo region would have to use a 4-byte Echo instruction for Alpha code. Furthermore, we may use a 5-byte Echo instruction to compress the Echo regions, which may be impossible for Alpha code.

Our result also suggests that ET compares favorably to the traditional procedural abstraction approach [Kunc-99]. Procedural abstraction converts common code sequences into separate procedures and uses the normal function call/ret to invoke the procedures. This approach has several disadvantages compared to ET. First, the reuse of sub-range within the echo target is now impossible. Second, Echo target is in the original code while procedural abstraction needs to use new code for both the target and the repeating code. Third, each "call" instruction takes 5 bytes in IA32, and the "ret" and stack manipulation operations take additional instructions, and this is much longer than an Echo instruction (2 to 4 bytes in Echo.I). [Kunc-99] shows that the Procedural Abstraction reduces the static code size of aggressively optimized SPEC96INT programs

by 0.85% to 2.37%. This is significantly less than that achieved by ET on already-size-optimized programs.

7 Future enhancement to ET for IA32

The major barrier to achieve higher code size reduction with ET is that the ECHO instruction needs to specify the offset from the Echo instruction to the Echo target. It is easy to see that when more code is searched, more Echo opportunities can be found. The Echo opportunities discovered via wide range searches may require a larger offset field in the Echo instructions. The larger offset requires longer versions of the Echo instruction, which may lose Echo opportunities when the Echo instruction itself is not shorter than the Echo region.

For a set of Echo instructions with long offsets, say, o1, o2, o3, …, ot, we may subtract a common "base" value from them all so that the new (delta) offsets o1-base, o2-base, …, ot-base are all much smaller than the original value. To compensate for the difference between the old offsets and the new offsets, we insert a new instruction "setEchoBase base" that will be executed before any of the Echo instructions dominated by it. This instruction will place the "base" value in a hardware "base_offset" register. When an Echo(offset, length) instruction is executed, the actual offset will be base_offset + offset, and this updated offset will be used to perform the Echo operation. We call this technique the *boosted Echo* Technology. We are currently evaluating its effects.

Currently we only allow call/ret/branch instruction with the same target address as that in the Echo target to be included inside the Echo region. It should be possible to allow them even when they have different absolute addresses in the Echo region and the Echo target. To compensate the address difference, the Echo instruction may take an additional operand for the difference to be added to the branch target address.

As proposed in [Brisk-02], recognizing Echo regions at the data flow level can potentially obtain much higher code size reduction. The challenge is to keep Echo region correctly maintained throughout optimizations after they are formed. This requires significant compiler changes. We will explore this avenue in the future.

Profile information may be used to guide the Echo algorithm to recognize Echo regions only in the infrequently executed code to reduce the performance impact of code compression [Krish-02]. Although performance critical code should not be recognized as Echo regions and replace them with Echo instructions, they still can be used as the Echo targets. In fact, allowing performance critical code to be Echo targets may enable the code to be more likely found in instruction cache and thus improve its performance.

We are also actively pursuing an efficient non-sequential search algorithm that may obtain better compression than our current sequential search algorithm. SEQUITUR [Manning-97] based search algorithm is interesting and it remains open whether or not we may adapt it to ET to out-perform the sequential search algorithm.

8 Conclusion

In this paper we first show that the current IA32 has code density disadvantage when compared to THUMB although its code density is much better than ARM. We then show that IA32 equipped with ET can achieve similar code density as THUMB, and there are opportunities to reduce the code size of IA32 programs further with techniques such as boosted Echo. Coupled with results reported in the literature and our preliminary evaluations, we conclude that IA32 with ET could incur significantly less performance loss than THUMB to achieve the similar code density. For IA32 targeting memory constrained embedded systems, we believe ET presents an attractive new technology.

9 Acknowledgements

We would like to thank Jeremy Lau and Philip Brisk for insightful technical exchanges about Echo technology, Tony Baker and Terry Smith for support and marketing research, Weidong Li and his team for helpful information on building and running Xscale compiler, and our colleagues at Programming Systems Lab for many helpful discussions.

References

[ARM-04] ARM website, e.g., http://www.arm.com /products/CPUs/archi-thumb2.html, 2004

[Brisk-02] Philip Brisk and Majid Sarrafzadeh, "Framework and Design Methodology of a Compiler that Compresses Code using Echo Instructions," ODES-2: 2nd Workshop on Optimizations for DSP and Embedded Systems, in conjunction with CGO04, March 21, 2004

[Cooper-99] K. D. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors," Proceedings of the Conference on Programming Language Design and Implementation, May 1999.

[CSIBE-04] GCC Code-Size Benchmark Environment (CSiBE), http://sed.inf.u-szeged.hu/csibe/ 2004

[Debray-02] S. Debray, W. Evans, R. Muth, and B. de Sutter. "Compiler techniques for code compression," ACM Trans. on Programming Languages and Systems, pages 378–415, 2000.

[Fraser-02] C. Fraser. "An instruction for direct interpretation of LZ77-compressed programs," Microsoft Technical Report MSR-TR-2002-90. ftp://ftp.research.microsoft.com/pub/tr/tr-2002-90.pdf.

[Intel-04] Embedded Intel® Architecture at http://www.intel.com/products/embedded/index.htm

[Krish-02] Krishnaswamy, Arvind, and Rajiv Gupta, “Profile Guided Selection of ARM and Thumb Instructions,” LCTES’02-SCOPES’02, June 19-21, 2002, Berlin, Germany, pp 56-64.

[Kunc-99] K. Kunchithapadam and J. Larus. “Using lightweight procedures to improve instruction cache performance,” CS-TR-99-1390, University of Wisconsin, 1999.

[Lau-03] Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, Brad Calder, “Code compression: Reducing code size with echo instructions,” October 2003 Proceedings of the international conference on Compilers, architectures and synthesis for embedded systems.

[Liao -96] S. Liao. “Code Generation and Optimization for Embedded Digital Signal Processors,” Ph.D. thesis, 1996. Massachusetts Institute of Technology.

[Manning-97] Nevill-Manning, C.G. and Witten, I.H. "Identifying Hierarchical Structure in Sequences: A linear-time algorithm," Journal of Artificial Intelligence Research, 7, 67-82. (1997)

[Nanja-04] Murthi Nanja and Joel D. Munter, “The Effects of Compiler Optimizations and Mixed Mode Code on Application Performance, Memory Footprint, Power, and Energy Consumption for Embedded Systems,” Submitted to CTCES04.

[NS-04] National Geode x86 "appliance-on-chip" SOCs, http://www.linuxdevices.com/products/PD6094486551.html

[VIA-04] Via Embedded Platforms at http://www.viaembedded.com/index.jsp.

[Ziv-77] J. Ziv and A Lempel, “A Universal Algorithm for Sequential Data Compression,” IEEE Transaction on Information Theory, 23 (3), p337-343, May 1977.